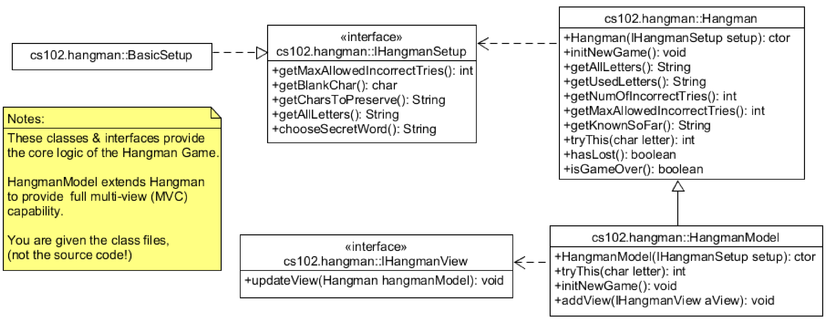
A GUI for the Hangman Game

In this lab you will create a GUI version of the hangman game. Your solution will use the MVC (Model-View-Controller) design pattern. This pattern separates the core logic of the game (the model), from the way in which the user interacts with it--how the user views and controls it. There are literally hundreds of variations of this basic theme. This lab introduces you to one that is (hopefully) easy to understand, yet principled and extensible. This should allow you to write a simple program to begin with and later, with minimum changes/effort, adapt it to much more sophisticated situations.

Part (a) - Get ready to play

Start by downloading and extracting this file into a new workspace folder for "lab05". The file contains a DrJava project with all the Java source and class files necessary to get you started. There is also a *readme.txt* file that explains what each sub folder contains... read it!

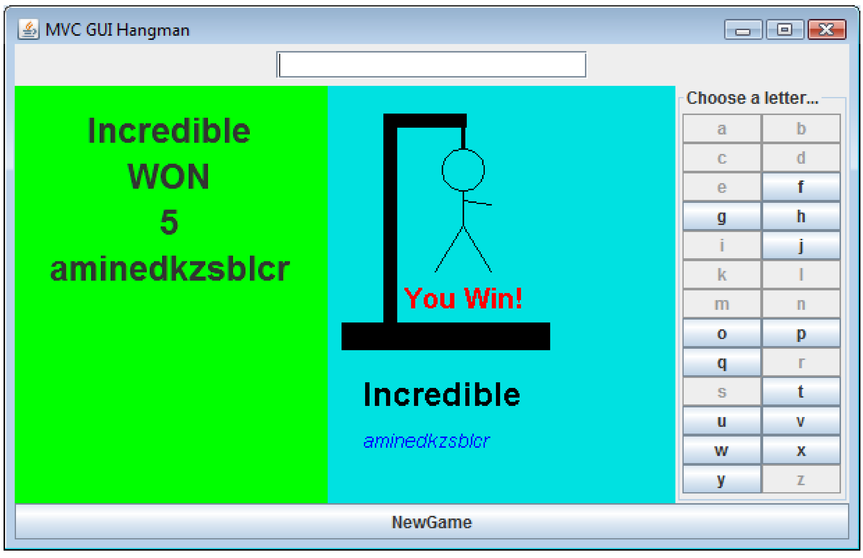
You should now see several folders, in particular "cs102", which contains the "**cs102.hangman**" package. This package contains the core hangman game classes (the *model*) and some interfaces, as shown in the UML class diagram below. Notice that the download includes only the *.class* (bytecode) files for these, not the source code. ***Important***: *You can put the "cs102" folder anywhere, but it must be on the classpath, otherwise you won't be able to compile the other classes!*



The "***src***" folder contains the "**hangmangame**" package, which is the real starting point. It contains **ConsoleHangmanDemo**, which has a main method that creates an instance of HangmanModel with a BasicSetup, adds an instance of HangmanConsoleView to it, then calls the *tryThis(letter)* method several times to check that it is all working properly. Try it... you should see the output appear on the Java console. The "hangmangame" package also contains **GUIHangmanDemo**, which is what we are about to build.

Everything else, including views and controllers, will be in the "**hangmangame.extras**" package. It currently has the HangmanConsoleView (which was used in the ConsoleHangmanDemo), plus an outline HangmanGUIPanel class, and one other file, LetterButtonControls, that we will use later! Most of the classes you will write should go into this package/folder.

Assuming all is well, we can begin on the GUI version. We will do this step-by-step, creating a collection of small components, including multiple views & controllers, which can be wired together as desired (to produce something like the example below).



Ok, it may not look very pretty, but it is functional and, perhaps more importantly, it is open to change. Take a look at the source code of the GUIHangmanDemo and the HangmanGUIPanel. The main method creates an instance of HangmanModel using an instance of BasicSetup. An instance of ConsoleHangmanView is added as a view of the model, as before (you can remove this line later, once the GUI is working). Following this, an instance of the HangmanGUIPanel is created (with the model as a parameter), and then added to the JFrame, which is then made visible as usual. This should all compile and run. Try it... (of course, there's nothing much to see, but it does provide a platform upon which to build), so let's get to work.

Part (b) - Adding controls.

We will first add some simple controls to the hangman game. In the "hangmangame.extras" package, create a new class, ***TextFieldControlPanel***, which extends JPanel. Its constructor should take a reference to a *Hangman* object and simply store it in a property. The class should also have a JTextField property, again initialised in the constructor. Add an ActionListener to the TextField. The listener (event handler) should get the text from the TextField and, for each letter in the text, call the *Hangman* (model) object's *tryThis* method. After processing all the letters it should clear the TextField. ***Note****: it doesn't matter how you make the listener; it can be an anonymous inner class, a named inner class, or the TextFieldControlPanel might itself implement ActionListener.*

In HangmanGUIPanel, create an instance of your new *TextFieldControlPanel* class (passing it the *HangmanModel*) and add it to the "north" the panel. Compile and run the project. You should now be able to type lots of letters into the TextField, press enter and they will be processed by the *HangmanModel* class, each time updating the *ConsoleHangmanView*. You might like to try adding another instance of your *TextFieldControlPanel* either to the "center" or the "south" of the *GUIHangmanPanel* and confirm you can control your hangman from either of them.

Next, create another class, ***NewGameButtonControl***. This time simply extend JButton directly. In the constructor, set the button's text to "New Game" and add an ActionListener which calls the *HangmanModel*'s *initNewGame()* method. Add an instance of the *NewGameButtonControl* class to the "south" of your *GUIHangmanPanel*. Confirm that it does indeed start a new game (i.e. chooses a new secret word) each time it is pressed.

Part (c) - Adding (GUI) views

It's now time to make some GUI views to replace the *ConsoleHangmanView* we have been using so far. Again, in the "hangmangame.extras" package, create a new class, ***LabelsHangmanView,*** which extends JPanel and implements *IHangmanView*. It will need JLabel's for each item you wish to show, e.g. *getNumOfIncorrectTries()*, *getKnownSoFar()*, *getUsedLetters()* and *hasLost()*. Later, you can adjust the layout, fonts, background colours, etc., to suit your taste. More important, however, is to make the *updateView(Hangman)* method, query the *Hangman* object for the required information and update the JLabel's accordingly.

In the HangmanGUIPanel, create a new instance of your *LabelsHangmanView*, add it to the *HangmanModel* as a view, and place it in the "west" of your panel. When you run the program it should now automatically update this view as well as showing the information on the console.

You should now be able to make a more conventional looking hangman game. Create a new class, ***GallowsHangmanView***, which also extends JPanel and implements *IHangmanView*. This time, however, you should override the JPanel's *paintComponent(Graphics)* method to display the gallows and the hanging man, based on the state of the *HangmanModel*--you can also include other information, e.g *getKnownSoFar()* and *getUsedLetters()* if you want. For this to work, your class will need to maintain a reference to the *HangmanModel* as a property, initialised via the constructor or set in the *updateView* method. The *updateView* method will normally simply tell the view to *repaint()* itself.

Create an instance of your *GallowsHangmanView* class, add it to the hangman model as a view, and place it in the center of the HangmanGUIPanel. Run the program and confirm it too shows the progress of the game.

Part (d) - Controls which are also views!

You may have noticed that the new game button is always enabled, allowing the user to abandon the current game and start a new one whenever they wish. While this may be acceptable, one might want to keep the button disabled by default, enabling it only after the current game has been properly completed. Doing this requires the button know the state of the game. The simplest way to do this is to make the control a view too, in other words, have the *NewGameButtonControl* implement *IHangmanView* too. Disable the button in its ActionListener, but in the *updateView* method if the game is over, re-enable it. Do these modifications and confirm the desired behaviour.

Finally, take a look at the *LetterButtonControls* class included in the project. This provides a simple virtual keyboard that can be used in lots of applications. Create an instance of it by passing it the letters you want to appear on the keys (obtained from the *getAllLetters()* method of the hangman object), and the number of *rows* and *columns* of buttons that you want (for example, try 13 & 2 to get a vertically oriented virtual keyboard for the 26 English letters). You can add it to the "east" of the HangmanGUIPanel, however, to have it work properly you will need to write an ActionListener and add it as a listener to *LetterButtonControls*. This time, do it by writing a separate listener class, *HangmanLetterButtonsController* say, that implements ActionListener. It should maintain a reference to the hangman model (passed in via its constructor). In the *actionPerformed* method it should get the letter of the button that was pressed and pass it to the *tryThis* method of the hangman object, then disable the button. ***Hint****: use the event source to decide which button*. Confirm that you can play the game using this keyboard.

There is one minor problem with this; if the user enters letters using the *TextFieldControlPanel*, the corresponding buttons on the ***LetterButtonControls*** are not disabled. This won't affect the logic of the game (entering the same letter again doesn't change anything), however, it is not very aesthetic. The difficulty only occurs when there are multiple controls some of which keep state. The solution is, again, to make the control a view too. Whilst you could modify the *LetterButtonControls* class, since these changes are specific to the Hangman class, it is probably best to leave it alone and make the necessary changes by sub-classing it. So, create ***HangmanLetterButtonControls*** which extends *LetterButtonControls* and implements *IHangmanView*. In the *updateView* method call the *setEnabledAll* & *setDisabled(String letters)* appropriately. ***Hint****: use the hangman model's getUsedLetters() method.*

~ Congratulations ~

You should have a working hangman game, but more importantly you should now have the knowledge and skills to engineer similar complex software systems in the future. We look forward to seeing what you can do. ☺

P.S. This is a work-in-progress... if you spot any errors or omissions, or have any suggestions for improvements to the assignment or the code, please do let us know.
